Expressions and Operators

Expressions

- Using variables within expressions to do something is what PHP is all about.

```
<?php
$name = 'Rob';
echo $name;
?>
```

Expression

Operator

Some Types of Operator

- Arithmetic
- Assignment
- Bitwise
- Comparison
- Ternary
- Incrementing /decrementing
- Logical
- String

String Operators

- Use a dot to concatenate two strings:

e.g.

```
$firstname = 'Rob';
$surname = 'Tuley';
// displays 'Rob Tuley'
echo $firstname.' '.$surname;
```

Arithmetic Operators

Example	Name	Result
`$a + $b`	Addition	Sum of `$a` and `$b`.
`$a - $b`	Subtraction	Difference of `$a` and `$b`.
`$a * $b`	Multiplication	Product of `$a` and `$b`.
`$a / $b`	Division	Quotient of `$a` and `$b`.
`$a % $b`	Modulus	Remainder of `$a` divided by `$b`.

Assignment Operators

Example	Result
`$a = $b`	Sets `$b` to the same value as `$a`.
`$a += $b`	Equivalent to `$a = $a + $b`.
`$a .= $b`	Equivalent to `$a = $a.$b`.

Combining Operators

- Note that you can combine operators, for example use =, + and / in one expression:

```
$a = 4;
$b = 2;
$c = $a + $b + ($a/$b);
// $c has value 4+2+(4/2) = 8
```

- Brackets help group operators.

Comparison Operators

Example	Name	Result
$a == $b	Equal	**TRUE** if **$a** is equal to **$b**.
$a != $b	Not equal	**TRUE** if **$a** is not equal to **$b**.
$a <> $b	Not equal	**TRUE** if **$a** is not equal to **$b**.
$a < $b	Less than	**TRUE** if **$a** is strictly less than **$b**.
$a > $b	Greater than	**TRUE** if **$a** is strictly greater than **$b**.
$a <= $b	Less than or equal to	**TRUE** if **$a** is less than or equal to **$b**.
$a >= $b	Gtr than or equal to	**TRUE** if **$a** is greater than or equal to **$b**.

Comparisons

- Comparison expressions return a value of TRUE (or ‘1’) or FALSE (or ‘0’).

e.g.

```
$a = 10;
$b = 13;
// result is true (‘1’)
echo $a < $b;
```

Incrementing/Decrementing

Example	Name	Effect
`++$a`	Pre-increment	Increments **$a** by one, then returns **$a**.
`$a++`	Post-increment	Returns **$a**, then increments **$a** by one.
`--$a`	Pre-decrement	Decrements **$a** by one, then returns **$a**.
`$a--`	Post-decrement	Returns **$a**, then decrements **$a** by one.

Logical Operators

Example	Name	Result
`$a and $b`	And	**TRUE** if both **$a** and **$b** are **TRUE**.
`$a or $b`	Or	**TRUE** if either **$a** or **$b** is **TRUE**.
`$a xor $b`	Xor	**TRUE** if either **$a** or **$b** is **TRUE**, but not both.
`!$a`	Not	**TRUE** if **$a** is not **TRUE**.
`$a && $b`	And	**TRUE** if both **$a** and **$b** are **TRUE**.
`$a || $b`	Or	**TRUE** if either **$a** or **$b** is **TRUE**.

Finally, a tricky one!

- A single **?** is the ternary operator.

```
(expr) ? if_expr_true : if_expr_false;
```

- A test expression evaluates to TRUE or FALSE.
  - **TRUE** gives first result (before colon)
  - **FALSE** gives second result (after colon)

Ternary Operator example

```
<?php
$a = 10;
$b = 13;
echo $a<$b ? ‘a smaller’:‘b smaller’;
// string ‘a smaller’ is echoed
// to the browser..
?>
```

Groups of variables

- So far, we have stored ONE piece of data in each variable.
- It is also possible to store multiple pieces of data in ONE variable by using an array.
- Each piece of data in an array has a key..

An array

Normal Variable, no key:

```
$name = 'Rob';
```

Array Variable, multiple pieces with 'keys':

```
$name[0] = 'Rob';
$name[1] = 'Si';
$name[2] = 'Sarah';
```

…

The 'key'

Array keys

- Array keys can be strings as well as numbers..

```
$surname['rob'] = 'Tuley';
$surname['si'] = 'Lewis';
```

- Notice the way that the key is specified, in square brackets following the variable name.

Working with arrays..

- Create Array (automatic keys):

```
$letters = array('a','b','c','d');
```

The array keys are automatically assigned by PHP as 0, 1, 2, 3

i.e. **`$letters[1]`** has value **`'b'`**

- Create Array (explicit keys):

```
$letters = array(10=>'a',13=>'b');
```

i.e. **`$letters[13]`** has value **`'b'`**

Working with arrays…

- Create array (component by component):

```
$letters[10] = ‘a’;
$letters[13] = ‘b’;
```

- Access array component:

```
echo $letters[10];
// displays a
echo $letters[10].$letters[13];
// displays ab
```

Working with arrays…

Note that trying to **`echo`** an entire array will not display the data. To print an entire array to screen (for debug, for example) use the function **`print_r`** instead.

~~`echo $letters;`~~

```
print_r($letters);
```

So..

We know we can:

1. Store things in named variables.
2. Use expressions to operate on the contents of these variables.
3. Can compare variables..

How do we actually include logic in the code such as ‘if this is bigger than that, do this’?

Control Structures

- **if, elseif, else**
- **while, do … while**
- **for, foreach**
- **switch**
- **break, continue, return**
- **require, include, require_once, include_once**

If …

- To do something depending on a comparison, use an if statement.

```
if (comparison) {
   expressions; // do if TRUE
}
```

- NB: Notice the curly brackets – these are important!

If example

```
<?php
$a = 10;
$b = 13;
if ($a<$b) {
   echo ‘a is smaller than b’;
}
?>
```

Extending IF statements

- It is possible to add extra optional clauses to if statements..

```
if (comparison) {
   expressions; // do if TRUE
} else {
   expressions; // do otherwise
}
```

Extending If statements

```
if (comparison1) {
   expressions;
} elseif (comparison2) {
   expressions;
} else {
   expressions;
}
```

An example..

```
$a = 10;
$b = 13;
if ($a<$b) {
     echo ‘a is smaller than b’;
} elseif ($a==$b) {
     echo ‘a is equal to b’;
} else {
     echo ‘a is bigger than b’;
}
```

While loops

- Might want to do something repeatedly while a comparison is true..

```
while (comparison) {
   expressions;
}
```

Example

- Lets count to 10! Displays 1,2,3,4,5,..,10:

```
$i = 1;
while ($i <= 10) {
    echo $i++;
}
```

Do .. While

- An alternative…

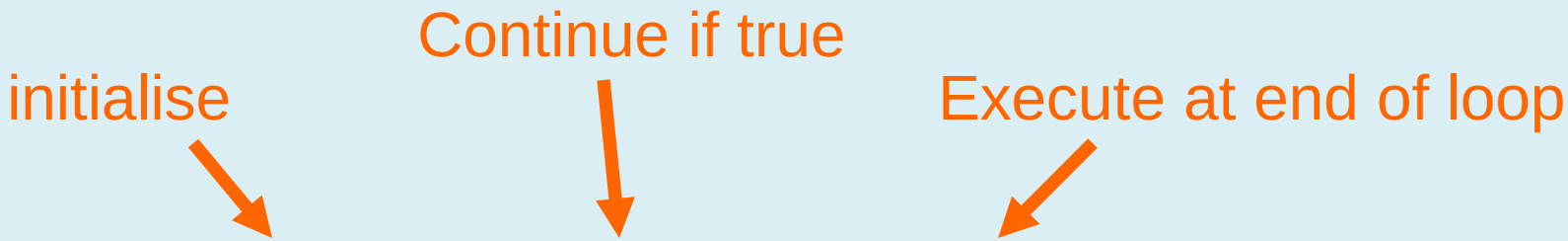
```
$i = 1;
do {
     echo $i++;
} while ($i <= 10);
```

For loop

- Sometimes we want to loop around the same bit of code a number of times.. Use a for loop.
- `for (expr1; expr2; expr3) { statements; }`
  - ***expr1*** evaluated/executed initially
  - ***expr2*** evaluated at beginning of each iteration (Continues if **TRUE**)
  - ***expr3*** evaluated/executed at end of each iteration

For loop example

- To count from 1 to 10:




```
for ($i=1; $i<=10; $i++) {
   echo $i;
}
```

Foreach loop

- A foreach loop is designed for arrays. Often you want to loop through each item in an array in turn..

```
$letters = array('a','b','c');
foreach ($letters as $value) {
    echo $value;
} // outputs a,b,c in turn
```

Foreach.. With keys

- Sometimes we want to use the array ‘key’ value too:

```
$letters = array(‘a’,’b’,’c’);
foreach ($letters as $key => $value) {
    echo “array $key to $value”;
}
```

Switch statement

- ***expr*** is evaluated
  - Case corresponding to result is executed
  - Otherwise default case is executed
- **break**
  - Ensures next case isn't executed

```
switch (expr) {
   case (result1):
     statements;
   break;
   case (result2):
     statements;
   break;
   default:
     statements;
}
```

Switch Example

```
switch ($name) {
   case ‘Rob’:
     echo ‘Your name is Rob’;
   break;
   case ‘Fred’:
     echo ‘You are called Fred’;
   break;
   default:
      echo ‘Not sure what your name is’;
}
```

break, continue, return

- **`break`**
  - Ends execution of current for, foreach, do … while, while or switch structure
  - Option: Number of nested structures to break out of
- **`continue`**
  - Skip rest of current loop
  - Option: Number of nested loops to skip
- **`return`**
  - Ends execution of current function/statement/script

Indentation..

- Code readability IS important – notice how all the code inside a loop/control structure is indented.
- Once you start writing nested control loops, indentation is the only way to keep track of your code!

require, include

- **`require('filename.ext')`**
  - Includes and evaluates the specified file
  - Error is *fatal* (will halt processing)
- **`include('filename.ext')`**
  - Includes and evaluates the specified file
  - Error is a *warning* (processing continues)
- **`require_once`** / **`include_once`**
  - If already included won't be included again

Code Re-use

- Often you will want to write a piece of code and re-use it several times (maybe within the same script, or maybe between different scripts).
- Functions are a very nice way to encapsulate such pieces of code..

Eh..? What?

- You have already used functions..

```
echo('text to display');
```

Function NAME

Function ARGUMENT

What is a function?

- A function takes some arguments (inputs) and does something with them (echo, for example, outputs the text input to the user).
- As well as the inbuilt PHP functions, we can define our own functions..

Definition vs. Calling

There are two distinct aspects to functions:

1. Definition: Before using a function, that function must be defined – i.e. what inputs does it need, and what does it do with them?
2. Calling: When you call a function, you actually execute the code in the function.

Function Definition

- A function accepts any number of input arguments, and returns a SINGLE value.

```
function myfunction($arg1,$arg2,…,$argN)
{
  statements;
  return $return_value;
}
```

Example

- Function to join first and last names together with a space..

```
function make_name($first,$last)
{
$fullname = $first.' '.$last;
return $fullname;
}
```

Calling functions..

- Can be done anywhere..

***myfunction*($*arg1*,$*arg2*,…,$*argN*)**

or

$answer = *myfunction*($*arg1*,$*arg2*,…,$*argN*)

e.g.

```
echo make_name('Rob','Tuley');
// echoes 'Rob Tuley'
```

Functions: Return Values

- Use **`return()`**
  - Causes execution of function to cease
  - Control returns to calling script
- To return multiple values
  - Return an array
- If no value returned
  - **`NULL`**

‘Scope’

- A function executes within its own little protected bubble, or local scope.
- What does this mean? Its means that the function can’t ‘see’ any of the variables you have defined apart from those passed in as arguments..
- Each new function call starts a clean slate in terms of internal function variables.

In other words..

- Variables within a function
  - Are local to that function
    - Disappear when function execution ends
- Variables outside a function
  - Are not available within the function
    - Unless set as global
- Remembering variables
  - Not stored between function calls
    - Unless set as static

Global variables..

- To access a variable outside the ‘local’ scope of a function, declare it as a global:

```
function add5toa()
{
  global $a;
  $a = $a  + 5;
}
$a = 9;
add5toa();
echo $a; // 14
```

Static Variables

- Local function variable values are not saved between function calls unless they are declared as static:

```
function counter()
{
  static $num = 0;
  return ++$num;
}
echo counter(); // 1
echo counter(); // 2
echo counter(); // 3
```

Default Arguments

- Can specify a default value in the function definition which is used only if no value is passed to the function when called..
- Defaults must be specified last in the list

```
function myfunction($arg1,$arg2='blah')…
```

~~`function myfunction($arg1='blah',$arg2)…`~~

Passing References

- Pass a reference to a variable
  - Not the actual variable
- Why?
  - Enables a function to modify its arguments
- How?
  - Use an ampersand in front of the variable
  - `&$variable`